N88-17263

COMMUNICATION AND CONTROL IN AN INTEGRATED MANUFACTURING SYSTEM[†]

Kang G. Shin, Robert D. Throne, and Yogesh K. Muthuswamy

Real-Time Computing Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109-2122

Abstract

Typically, components in a manufacturing system are all centrally controlled. Due to possible communication bottlenecking, unreliability, and inflexibility caused by using a centralized controller, a new concept of system integration called an *Integrated Multi-Robot System* (IMRS) was developed. The IMRS can be viewed as a distributed real-time system.

This paper presents some of the current research issues being examined to extend the framework of the IMRS to meet its performance goals. These issues include the use of communication coprocessors to enhance performance, the distribution of tasks and the methods of providing fault-tolerance in the IMRS. An application example of real-time collision detection (as it relates to the IMRS concept) is also presented and discussed.

1 Introduction

Conventionally, components in a manufacturing system are all centrally controlled; that is, control tasks for the system may be distributed over a network of processors or reside in a uniprocessor but are all executed under directives of one central task. The work by Maimon [1] [2] is primarily concerned with dynamically determining how to utilize the resources within a workcell to achieve a certain objective, where an activity controller provides for centralized control of the workcell. The work at the National Bureau of Standards on their Automated Manufacturing Research Facility (AMRF) system [3] [4] [5] deals with real-time control of a workcell using strictly hierarchical control. Their system is data-driven and based on state tables at each level of hierarchy. At each level, these state tables are updated on the basis of (1) commands from the next higher level, (2) results of processes at the next lower level, and (3) sensor inputs at the current level. While information can be exchanged across one level, control is strictly vertical. The state table approach allows for recovery from various undesirable events (so long as these events are accounted for in one of the states), but the overall sequence of operations is hidden from the user.

Due to possible communication bottlenecking, unreliability, and inflexibility caused by using a central controller, we have proposed a new concept of system integration, called an *Integrated Multi-Robot System* (IMRS) [6] [7]. An IMRS is defined as a collection of robots, sensors, computers, and other computer controlled machinery, such that

- each robot is controlled by its own set of dedicated tasks, which communicate to allow synchronization and concurrency between robot processes,
- tasks execute in parallel,
- both centralized and decentralized control concepts are used, and
- tasks may be used for controlling other machinery, sensor I/O processing, communication handling, or just plain computations.

In the above definition (and in what follows) the term "process" refers to an *industrial* (but not computational) process, which could be decomposed into several *subprocesses*. Each subprocess may be accomplished by executing a software *module* in a computerized controller. Each module can be decomposed further into computational *tasks*.

The goal of an IMRS is to outperform its counterparts by better utilization of physical space and computer capabilities, increased throughput, greater flexibility, improved fault-tolerance, and the capability of handling diverse manufacturing processes. In order for an IMRS to effectively utilize the available resources, it must make maximum use of the possible parallelism between processes and tasks. In an IMRS there are five different classes of interaction between subprocesses[7]:

- *Independent Processes*: the work of each subprocess is independent, and the actions taken by each subprocess

[†] This work was supported in part by the NASA Johnson Space Center under Grant No. NCC–9–16 and the US Airforce Office of Scientific Research under Contract No. F33615–85–C–5105.

PRECEDING PAGE BLANK NOT FILMED

to accomplish its goal are also independent. Indirect influence through state variables is the only way the subprocesses of an independent process may be related.

- *Loosely Coupled Processes*: the subprocesses perform independent work, but the actions taken by each subprocess depend on the actions of the other subprocesses, e.g., two robots sharing the same workspace or set of tools.
- *Tightly Coupled Processes*: the work of the subprocesses depend on each other, and the actions taken to carry out subprocesses also depend on each other. Carrying a long steel beam with two robot arms is a typical example of this class.
- *Serialized Motion Processes*: the works of the subprocesses depend on each other, yet the actions taken to accomplish each subprocess are independent, e.g., assembly.
- *Work Coupled Processes*: the processes monitor each other. Should one process crash due to a computer or device failure, the other computer or device will attempt to take over the responsibilities of the failing device or computer.

Using the above classification, a logical communication architecture called *module architecture* and those primitives necessary for an IMRS are identified in [7]. The module architecture for an IMRS is an n-ary tree that is formed by *task creation*. When a task is created, it becomes a child task of the task that created it. This parent/child relationship always exists, but the amount of communication between the two will be different according to the class of process the tasks are controlling. Under most circumstances, communication channels among child tasks will be directly established, with the parent task playing a minor role. This is defined as *horizontal communications*. However, in some cases the parent must tightly control its child tasks. This is defined as *vertical communications*. Note that these two approaches represent decentralized and centralized controls, respectively. A *proprietor* or *administrator* task is used to provide exclusive access to shared resources (e.g., the right to change a state variable) and resolve conflicts among different concurrent tasks.

We assume that processors controlling devices in one workcell communicate over a common bus or a local area network[1], while GM's Manufacturing Automation Protocol (MAP) [8] is used for communication between workcells. MAP is a protocol for local area networks based on the OSI (Open Systems Interconnection) Reference Model developed by ISO and CCITT. It is a seven layer communication protocol which uses a token passing bus based on the IEEE 802.4 standard [9] [10] as the physical layer. The application layer of MAP specifies the use of the Manufacturing Message Specification (MMS) [11] [12] for communication with manufacturing and process control devices. For time critical applications, the upper four layers of the seven layer ISO protocol are removed, leaving a three layer protocol called the *MiniMAP*. Thus, MiniMAP does not conform to the OSI standard since it is incapable of peer open system communication. MiniMAP is suitable for unintelligent devices such as sensors that do not need to communicate outside their interconnection network. MAP/EPA is composed of both the full seven layer MAP and the three layer MiniMAP.

In the context of the IMRS concept, we discuss in subsequent sections various issues in system integration, such as architectures for high performance intertask communications, the distribution of device controllers among the networked computers, and graceful degradation in case computers and devices fail. Communication bottlenecks should be avoided with any distributed system, particularly with a real-time system. In addition, the assignment and scheduling of tasks on a processor in such a system is of paramount importance. In order to minimize communication bottlenecks and allow for real-time task management, the use of a communication coprocessor is discussed in Section 2. The distribution of tasks on a distributed system has been studied previously. Section 3 discusses some of the issues involved in task distribution in a real-time control environment like an IMRS. Section 4 discusses fault-tolerance in the IMRS, particularly the problems with work coupled processes. Section 5 discusses an application of the IMRS concept to real-time collision detection and avoidance. Finally, Section 6 summarizes the paper.

2 IMRS Communications

An IMRS can be considered a distributed real-time system, with each of the workcells considered as a node. A workcell refers to a set of processes which are grouped together either due to their functional relationship or due to physical proximity of the devices they use. Communications within the same workcell are usually more intense and time-constrained than the communications taking place between different workcells.

We will consider the use of a port-based communication architecture for the IMRS because of its many advantages such as modularity, flexibility, and programmability (see [7] for more on these). In this architecture, each task is associated with some ports to communicate with other tasks. These ports are logical entities and may be mapped onto physical ports on processor nodes on which their associated tasks are located. It is natural to decompose each node's function of the IMRS into communications and applications. For the high performance required for the IMRS, the former will be handled by a dedicated processor called a *communication processor* (CP) and the latter by an *application processor* (AP). The idea of using hardware support for interprocess communication has been proposed elsewhere [13], though not in the context of real-time control. The AP may either be one physical processor or multiple processors. The CP is responsible for all the

[1]That is, a network consisting of only the processors and devices within one workcell.

communications associated with the tasks residing at the node and the AP is responsible for the necessary computation, e.g., execution of a robot's motion.

The processes within a workcell are accomplished by executing a set of tasks, possibly on different processors. A contemporary workcell consists of a number of coprocessors which execute different tasks and also has a CP which is responsible for communicating with the other workcells.

The inter-node access protocol will play a key role in the overall system performance. Notable among popular performance parameters are: *response time*, *throughput*, *availability*, and *fairness*. Response time is composed of nodal computation time at each layer, queueing delays at each layer and at each node, and the actual propagation time along the network. For example, with the IEEE 802.4 token passing scheme (used by GM's MAP), an upper bound exists for the time a node will have to wait to transmit some of its data. The throughput of the network basically depends on the buffering capacity of the destination node or of any intermediate nodes (i.e., gateways) the messages must pass through. Effective throughput is a function of the number of retransmissions required due to transmission errors. The availability of the network depends on the reliability of the components used in the network. For example, if the node with the token fails, the network will be unavailable until error recovery procedures reconfigure the system and generate a new token. Finally, the fairness of the network depends on the load demanded by each user and the optimization the network provider is trying to achieve. For example, if the network provider optimized mean response time in the network, then it is better to allow transmission of users' packets equally. On the other hand, if the network provider optimized throughput, then it is better to allow transmission of packets from users who have the maximum demand. For more information, see [14].

Unlike the inter-node communications, the organization and communications within a workcell node are determined by a number of other issues related to the *message handler* (MH). The MH is a task responsible for interfacing each task on the workcell with its environment. Each task is associated with a MH task, and the aggregation of all the MH tasks at a node resides in the CP and acts as the communication interface for all the tasks associated with that node. This aggregation will henceforth be referred to as MH for simplicity.

The tasks queue up their requests to the CP (either to send or receive messages) on independent queues. The MH task *scans* all these queues and selects a request to service based on some criteria, for example, priority of the requesting task, or the deadline associated with the message to be sent. The task priorities may either be determined *a priori* or dynamically. After sending a message, some tasks might get blocked. Also, when a message arrives from some other node, some tasks might get unblocked. When a task currently executing on the AP gets blocked after sending a message, the MH should decide the next task to be scheduled on the AP. Similarly, when a task gets unblocked, scheduling decisions have to be made by the MH. The methodology of scanning the requests by MH, and an appropriate algorithm to select a request to service will be an important research area.

In addition to acting as the interface for the tasks at a node, the MH is also responsible for maintaining the required degree of fault-tolerance. Failures might be due to device or processor failures. The MH maintains a task map at the node. It is also responsible for unblocking processes that have been blocked by the failure(s) of devices or processors.

The interconnection between the various processors at a workcell depends very much on the pattern and intensity of the communications taking place and also on the stringency of the deadlines associated with the various tasks. A possible interconnection is to connect all the processors in a ring. In this case the time for message passing between any two processors will not be the same, but this allows for expandability. The entire workcell can be visualized as a hierarchy of levels. At the lowest level we have the tasks and the message handlers associated with them. At the next level is the CP associated with that workcell node. The interconnection between the various levels and also the interconnections within the same level have to be determined. Another issue is whether to implement the MH in hardware or software, i.e., whether additional processing power should be provided to each task to implement the MH, or can it be done by the CP at the higher level. This would depend on the fault-tolerance sought for the system as well as the message traffic pattern and intensity.

The protocols used at various levels must be studied. A traditional seven layer protocol at the device controller level may result in deadlines being missed due to the time overhead involved. The sensitivity of the deadlines to various parameters like protocols and interconnection is an important issue and will determine the overall architecture.

3 Distributing IMRS Tasks Among Processors

The distribution of the tasks on the processors will be a key element in determining the overall system cost, performance, and reliability. By examining the parallelism between tasks we get some indication of which tasks can be assigned to the same processor without performance degradation. In addition to the classification of processes, one of the distinct features of an IMRS is to allow *both* vertical and horizontal communications. If the control tasks are distributed over many processors, a hybrid of horizontal and vertical communications between tasks may prove to be beneficial.

For example, serialized tasks can be assigned to the same processor, while assigning independent tasks to the same processor may result in a serious performance degradation. However, since some tasks may depend on state variables modified in another processor, delays in reliably updating these variables must also be included when assigning tasks to processors. If the network throughput is too low, assigning all tasks dependent on one or two key state variables to a single processor (even if the tasks are independent or loosely

coupled) will improve system performance. From these arguments it appears beneficial to group many tasks on a few large (and powerful) processors, but this could lead to a decrease in system performance and reliability.

The system throughput might increase if processors were physically located near the devices to be controlled, each processor having a direct access to the device (i.e., through an I/O port). In this way, a control task for a device could be assigned to its "local processor" and would have to contend with smaller delays over the physical network. There are, however, several drawbacks to this idea. If we depend on having a processor at each device, the potential reliability of networking the computers is seriously diminished. If our real-time performance depends on the presence of such processors, and a local processor fails, we may not be able to have another processor assume the control task and meet the real-time constraints. In addition, if the device only communicates through the local processor's I/O ports, and the processor fails, we may not be able to communicate with a (working) device.

Allowing tasks to communicate directly (horizontally) without a central controller (i) reduces the chances of a bottleneck by exchanging messages among children (instead of always going through the parent), (ii) increases reliability because the subprocesses do not rely on one central control task, and (iii) allows more parallelism because each child task is not blocked as often as in the vertical case, where each child must always wait for a directive from the parent. Tradeoffs in using vertical and horizontal communications for various industrial processes must be analyzed.

Most methods for allocation of tasks in a distributed system are concerned with minimizing a cost function consisting of the sum of processing cost per task on each assigned processor and interprocessor communications (IPC). As was reviewed in [15], these methods are based on graph theory or integer programming or heuristic solutions. Real-time constraints are difficult to impose using the graph theoretic approach, while the integer programming methods allow constraints that all of the tasks assigned to a processor complete within a given time. However, this constraint does not account for task queueing and precedence relations among tasks.

Efe [16] presents a module clustering algorithm minimizing IPC cost without considering constraints, and then moves modules from overloaded to underloaded processors by a module reassignment algorithm. Ma *et. al.* [17] developed an algorithm based on integer programming and the branch-and-bound method. A *task exclusive* matrix defined mutually exclusive tasks that could not be placed on a single processor and *task redundancy* was introduced for system reliability. Chu and Lan [18] chose to minimize the maximum processor workload in the allocation of tasks in a distributed real-time system. Workload was defined as the sum of IPC and accumulated execution time for each processor. A *wait-time-ratio* between two assignments was defined in terms of the task queueing delays. Precedence relations were used to arrive at two heuristic rules for task assignment, which were used in conjunction with the wait-time-ratios to generate a heuristic algorithm for task allocation. Lo [19] proposed the concept of *interference costs* which were inferred when two tasks were assigned to the same processor. This additional cost was used in an effort to reward concurrency.[2]

A criterion to measure task assignments in the IMRS using some of the ideas mentioned above must be developed. It should include task redundancy and mutual exclusion to provide reliability, as well as requirements to group certain tasks to be executed on a single processor. An IMRS should take advantage of as much parallelism as possible, so we will need to include some type of interference penalty. Since we have to deal with a real-time system, we will need to account for queueing delays in the network and within a processor. Finally, the IMRS deals with five basic task classes, and the cost function will have to deal with tasks within the different classes separately.

Once an appropriate cost function is determined, an algorithm to distribute the tasks to the processors must be developed. It is unlikely that a polynomial time algorithm will be found, so faster heuristic suboptimal algorithms may have to be developed.

4 Fault-Tolerance

One of the primary reasons for using a distributed system is to improve the fault-tolerance of the system. The IMRS deals with fault-tolerance through work coupled processes or tasks. These tasks monitor each other so that if the processor or device executing one task fails, the other task on the healthy processor can attempt to compensate. In order to compensate for tasks on a failed processor, the states of those tasks must be known. The update rate between work coupled tasks will affect both network traffic and the load of the associated message handlers. If the state of each work coupled task is updated too often, the network may get congested with state update messages, while if the state is not updated often enough, then recovery of the failed process will be more difficult. Finally, work coupled tasks should not be assigned to the same processor, since failure of that processor will make recovery impossible.

For work coupled tasks to be effective, the system must have the ability to determine that a processor or device has failed. Hence we must first determine methods of detecting the failure of a processor.[3] One such method is sending *heartbeat* messages between processors and assuming the failure of a processor if a response is not received within a prescribed time. A critical issue is the number of such messages and the rate at which they are sent. Depending on the system architecture and timing constraints, it may prove beneficial to have such heartbeat messages sent at different rates for

[2]That is, the assignment of two tasks which could be run simultaneously if assigned to different processors would tend to produce a lower objective function if such an assignment were made.

[3]We assume a *fail stop* system, where a processor stops when it fails.

different processors. These rates would be determined by (1) the minimum allowable recovery time of any of the tasks on the failed processor, (2) the minimum state update rate of any of the tasks on the failed processor, and (3) the assignment of tasks to processors. In addition, the destination of each heartbeat message will depend on these factors, since heartbeat messages could also serve as state update messages.

It may be useful to have specific *health management* tasks to maintain system health, rather than having individual processors acting independently. For example, the health management tasks would be responsible for initiating all heartbeat messages, maintaining tables of healthy processors and the tasks running on those processors, and coordinating recovery when a processor failed. While we may be able to save time and resources by having a single health management task, these benefits would have to be realized at the expense of a centralized system. We would certainly want to have redundant copies of the health management tasks, and may want to have two or more copies of the same task running simultaneously on different processors.

Once a processor is determined to have failed, we must devise mechanisms for ensuring that none of the tasks on the working processors remains "blocked" while waiting for a reply from a task on the failed processor. To accomplish this we can have the message handler maintain lists of incoming and outgoing messages and issue "fake" messages [20] to the blocked tasks. In addition, since the IMRS communicates through ports, the lists of users of a port must be updated to reflect the current state of the system. If a task realizes that one of the work coupled tasks it is monitoring has failed, it should assume that task. Should it then also try to set up a new work coupled task to monitor itself on another processor? One solution might be to have many "overlapping" work coupled tasks assigned when the system is initialized. However, the extra network traffic caused by this solution could be high. Instead, we could have a hierarchical system of work coupled tasks, in which states are updated less often at lower levels of the hierarchy. Such a system for establishing checkpoints in order to achieve resiliency was proposed in [21]. In addition, we would have to determine how many overlapping work coupled tasks would provide the desired degree of fault-tolerance. Similarly, suppose a processor fails and all of the tasks executing on it are assumed by other processors. Now the first processor is restarted. We need to determine a mechanism to dynamically reassign tasks to the processor when it is restarted. Certainly, we do not want to have to shut down the network (and hence the manufacturing) just to reload one processor. The requirements for such a system are presented in [22]. In case there is a sufficient number of failures that not all of the tasks can be run in real-time, these tasks must be executed in a preplanned degraded mode.

5 Real-Time Collision Detection in an IMRS

To discuss their feasibility, the IMRS concepts and solutions must be applied to some realistic examples. Due to its importance, real-time obstacle detection and avoidance has been selected as an application example.[4] This example requires the IMRS to communicate effectively with external sensors, such as vision systems, acoustic range sensors, and various types of proximity sensors. To maintain a high degree of fault-tolerance, each of these sensors should be linked to the computer network. We expect the sensors to provide overlapping coverage, so that if some of the sensors fail information from the other sensors can be used to continue.

Initially, we will assume that the "obstacles" are AGVs conveying parts between workcells. We do not want all devices on the factory floor to stop whenever an AGV nears a device or workcell, only those workcells and devices which potentially could collide with the AGV should be stopped or slowed. Define a *workcell safety volume* as the volume enclosing the workcell which cannot be safely entered while the devices in the workcell continue normal operation. Note that it may be possible to safely enter a workcell safety volume if the devices within a workcell are slowed down or their operations are changed. A *device safety volume* is similarly defined as the volume surrounding a device which cannot be safely entered while the device remains in normal operation.

Associated with the notion of these safety volumes, assume that there are two levels of collision detection, *workcell volume warning* and *device volume warning*. The former provides warning that with an obstacle's current trajectory[5] it may intersect a particular workcell's safety volume, or a group of workcells' safety volumes. This is early warning that the devices in the workcell may have to stop or otherwise alter their normal operation. Similarly, device volume warning provides warning that a particular device's safety volume, or a group of devices' safety volumes, may be violated. If a device's safety volume is violated, the device *must* take immediate actions to avoid a collision.

Define *collision detection* (CD) tasks as those tasks assigned to track obstacles and determine whether any safety volumes will be violated. These tasks must estimate the earliest violation of any device's or workcell's safety volume in terms of some parameter. In addition, since there may be many obstacles present in the environment, the CD tasks must determine, for each message received from the sensors, whether a current obstacle is one which it is already tracking, whether the current obstacle presents a threat to any of the devices or workcells the CD task is monitoring, or whether the obstacle is a new threat.

[4] However, we will address only those issues related to the IMRS.

[5] Actually, since we will be sampling at discrete times, we will probably include all possible trajectories between the current time and the next sample time. For example, we may assume the moving obstacle can instantaneously change direction, and will search for all intersections within a given radius around the obstacle.

We define *device stopping* (DS) tasks as those tasks which determine how a device (or group of devices) can safely stop and how long (in terms of some parameter) the device (or group of devices) require to stop. For example, if two robots are carrying a heavy panel it may not be safe to have each robot just stop as quickly as they can (individually). This uncoordinated action may cause them to drop the panel or even damage themselves. Instead, we may want them to stop as fast as possible while not deviating from their preplanned path (to avoid any further collisions).

An important issue is how many CD and DS tasks should there be, and what their relationships should be with the other tasks. One option would be for each workcell to have its own CD task, and if an obstacle comes within a prescribed minimum distance, the CD task would spawn subtasks for the individual devices within the workcell. However, the overhead associated with setting up new tasks may be prohibitive. Also, we may not know that there is enough computing power available to run each of these tasks in real-time. A better idea might be for these tasks to be preassigned to processors but remain "inactive" until required. As more processing by the CD tasks is required, the other tasks would be forced to slow down. Since we would probably want the devices to slow as an obstacle came near, this may not be much of a problem if the CD tasks and the device controlling tasks were assigned to the same processor.[6]

Another issue here is in dynamic priority assignments. As an obstacle comes near, we may want the CD tasks to have the highest priority. When a collision becomes imminent we want the task controlling the stoppage of a device to have the highest priority (and not be interruptible).

6 Summary

We are currently investigating various issues of system integration, the solutions of which will extend the framework of an IMRS to meet its real-time performance and fault-tolerance goals. While many of the issues presented are currently being studied in the literature, few solutions deal with the special requirements of the IMRS. The use of a communication coprocessor to speed up communications, provide real-time task scheduling, and maintain tables and lists for fault-tolerance has been discussed. Issues related to the task distribution in an IMRS have been addressed. The use of work coupled tasks to recover from failed tasks, as well as the use of heartbeat messages to determine failed processors has been examined. The solutions of these problems will benefit not only the IMRS, but also other distributed real-time systems.

[6]There may be a problem if, for example, robots were following a prescribed trajectory. In this case, we may not be able to follow the trajectory without sufficient computational power. We may be able to follow the same path, though.

References

[1] Maimon, O. Z., and Nof, S. Y., "Coordination of robots sharing assembly tasks", *Journal of Dynamic Systems, Measurement, and Control*, Vol. 107, December 1985.

[2] Maimon, O. Z., "A multi-robot control experimental system with random parts arrival", *IEEE Conference on Robotics and Automation*, St. Louis, MO, March 1985.

[3] Simpson, J. A., Hocken, R. J., and Albus, J. S., "The automated manufacturing research facility of the National Bureau of Standards", *Journal of Manufacturing Systems*, Vol. 1, No. 1, 1984, pp. 17-31.

[4] Jones, A. T., and McLean, C. R., "A proposed hierarchical control model for automated manufacturing systems", *Journal of Manufacturing Systems*, Vol. 5, No. 1, pp. 15-25.

[5] Haynes, L. S., Barbera, A. J., Albus, J. S., Fitzgerald, M. L., and McCain, H. G., "An application example of the NBS robot control system", *Robotics and Computer-Integrated Manufacturing*, Vol. 1, No. 1, 1984, pp. 81-95.

[6] Shin, K. G., Epstein, M. E., and Volz, R. A., "A module architecture for an integrated multi-robot system", *Technical Report*, RSD–TR–10–84, Robot Systems Division, Center for Research and Integrated Manufacturing (CRIM), The University of Michigan, Ann Arbor, MI, July 1984. Also appeared in the *Proc. 18th Hawaii Int'l Conf. on System Sciences*, January 1985, pp. 120-129.

[7] Shin, K. G., and Epstein, M. E., "Intertask communications in an integrated multi-robot system", *Technical Report*, RSD–TR–4–85, Robot Systems Division, Center for Research and Integrated Manufacturing (CRIM), The University of Michigan, Ann Arbor, MI, May 1985. Also appeared in *IEEE Journal on Robotics and Automation*, Vol. RA-3, No. 2, April 1987, pp. 90-100.

[8] Manufacturing Automation Protocol (MAP) Reference Specification (Draft), February 25, 1986.

[9] IEEE Standards Board. IEEE Standards for Local Area Networks: Token-Passing Bus Access Method and Physical Layer Specification. New York: IEEE, 1985.

[10] Stallings, W., "IEEE Project 802 : Setting standards for local-area networks", *ComputerWorld*, February 1984.

[11] "Manufacturing Message Specification – Part 1: Service Specification", *ISO 2nd DP 9506*, May 21, 1987.

[12] "Manufacturing Message Specification – Part 2: Protocol Specification", *ISO 2nd DP 9506*, May 21, 1987.

[13] Ramachandran, U., "Hardware support for interprocess communication", *Computer Sciences Technical Report # 667*, University of Wisconsin, Madison, WI., September 1986.

[14] Muralidhar, K. H., "Performance management – measures, analysis, control, and optimization", *Proc. 11-th Conference on Local Computer Networks*, October 1986, pp. 20-25.

[15] Chu, W. W., Holloway, L. J., Lan, M. T., and Efe, K., "Task allocation in distributed data processing", *Computer*, November 1980, pp. 57-69.

[16] Efe, K., "Heuristic models of task assignment scheduling in distributed systems", *Computer*, June 1982, pp. 50-56.

[17] Ma, P. Y. R., Lee, E. Y. S., and Tsuchiya, M., "A task allocation model for distributed computing systems", *IEEE Transactions on Computers*, Vol. C-31, No. 1, January 1982, pp. 41-47.

[18] Chu, W. W., and Lan, L. M. T., "Task allocation and precedence relations for distributed real-time systems", *IEEE Transactions on Computers*, Vol. C-36, No. 6, June 1987, pp. 667-679.

[19] Lo, V. M., "Heuristic algorithm for task assignment in distributed systems", *Proc. 4-th International Conference on Distributed Computing Systems*, May 1984, pp. 30-39.

[20] Knight, J. C., and Urquhart, J. I. A., "On the implementation and use of Ada on fault-tolerant distributed systems", *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 5, May 1987, pp. 553-563.

[21] Birman, K. P., Joseph, T. A., Raeuchle, T., and Abbadi, A. E., "Implementing Fault-Tolerant Distributed Objects", *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 6, June 1985, pp. 502-508.

[22] Kramer, J. and Magee, J., "Dynamic Configuration for Distributed Systems", *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 4, April 1985, pp. 424-435.